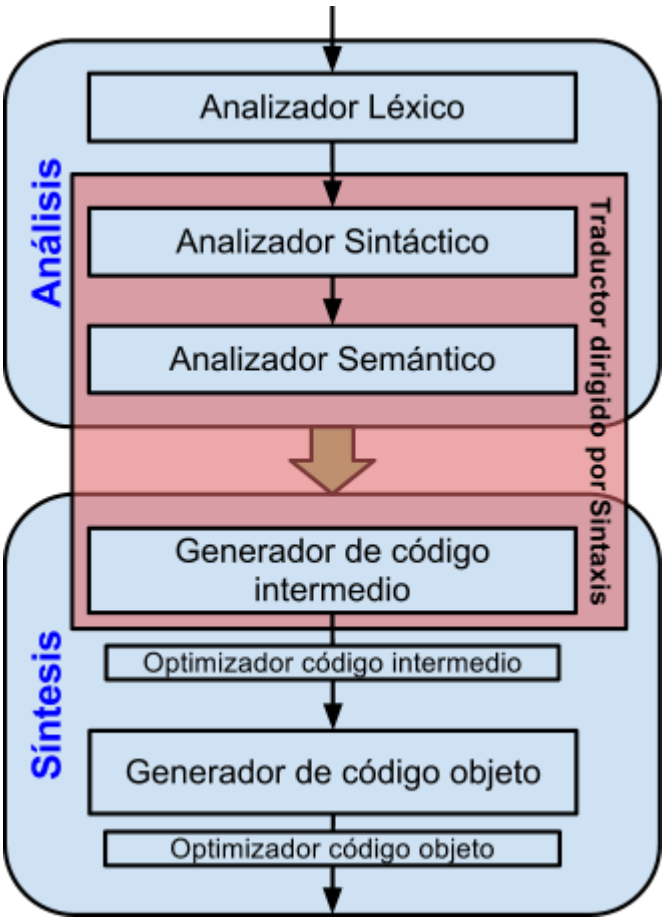
Traductores del lenguaje

Pau Arlandis, Andrés Orcajo, Guillermo Ramos, Álvaro J. Aragoneses.

1. Introducción



Se realiza la traducción en dos pasos, por dos razones:

- Portabilidad → El código intermedio puede ser único para muchos análisis. Además es independiente de la máquina.
- Sencillez → Es más fácil dar dos pequeños saltos que uno grande.

Podríamos decir que en procesadores del lenguaje el análisis sintáctico y el análisis semántico estaban unidos para que, al recibir *tokens* no fuese necesario crear un árbol y que el análisis semántico lo recorriese para crear una definición dirigida por la sintáxis. En esta asignatura vamos a ver que es posible unir el generador de código intermedio a estos dos analizadores en un único gran bloque que recibe *tokens* a la entrada y devuelve código intermedio. Esto se denomina Traducción dirigida por la sintáxis.

2. Generador de código intermedio

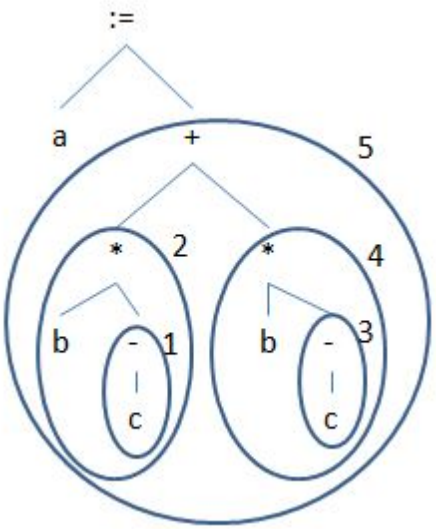
2.1. Lenguajes intermedios

El código intermedio podría ser de diferentes formas:

- Representación gráfica:
  - Árboles.
  - Grafos dirigido acíclico.
- Notación Polaca Inversa (RPN).
- Código de 3 direcciones .

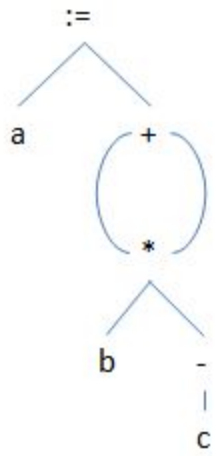
Como ejemplo utilizaremos la siguiente frase de la que suponemos el análisis correcto:
$a := b * - c + b * - c$

2.1.1. Árboles



Como vemos es importante conocer el orden de los operadores.
Podemos utilizar un grafo dirigido acíclico para eliminar redundancia.

2.1.2. GDA



2.1.3. Notación polaca inversa

RPN son sus siglas en inglés. Es una notación donde primero se colocan los operadores y después el código de operación que utiliza esos operadores. Por ejemplo:
$a+b \rightarrow ab+$
$a*b \rightarrow ab*$
$a*b+c \rightarrow ab*c+$
$a+b*c \rightarrow abc*+$
En nuestro ejemplo
$a:=b*-c+b*-c \rightarrow abc-*bc-*+:=$ En este caso - es un operador unario, se sabe porque el código es diferente.

De forma conceptual podemos ver la RPN como una linealización de un árbol.

Esta representación parece ideal pero solo con expresiones (sobre todo aritméticas) pero se torna ineficiente con cierto tipo de sentencias (if-then-else, do-while,...), muy comunes en un lenguaje de programación. Por ejemplo, en un if-then-else evalua la rama del Then y del Else antes de evaluar la sentencia if.
$if\ x>7\ then\ y:=6\ else\ y:=8 \rightarrow x7>y6:=y8:=IF$ Siendo IF el código de operación de la sentencia if-then-else, considerada en este caso una sentencia triaria.

Esta ineficiencia solo puede resolverse con el método que vamos a utilizar para el código intermedio: el código de 3 direcciones.

2.1.4. Código de 3 direcciones

Son instrucciones como máximo de tres direcciones (operador operando operando). Por ejemplo:

- $x:=y\ op\ z$

- $x := op\ y$

En el ejemplo

$a := b * - c + b * - c$

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

Como se ve, cada operación se hace a parte y se van uniendo, siempre en operaciones de 3 direcciones como máximo.

Ejercicio 1

Diseñar el Generador de Código Intermedio mediante Definición Dirigida por la Sintaxis (DDS) para esta gramática y que obtenga como salida el código de 3 direcciones (3-d):

1. S → id := E
2. E → E + E
3. E → E * E
4. E → -E
5. E → (E)
6. E → id

Utilizar los atributos:

- .lugar → Representa la posición [dirección de memoria o registro de la máquina] en la que está ese id (o resultado) en tiempo de ejecución.
- .cod [.código] → Contiene el conjunto de instrucciones en código de 3 direcciones correspondiente a la traducción de ese símbolo gramatical.

Cómo ejemplo:

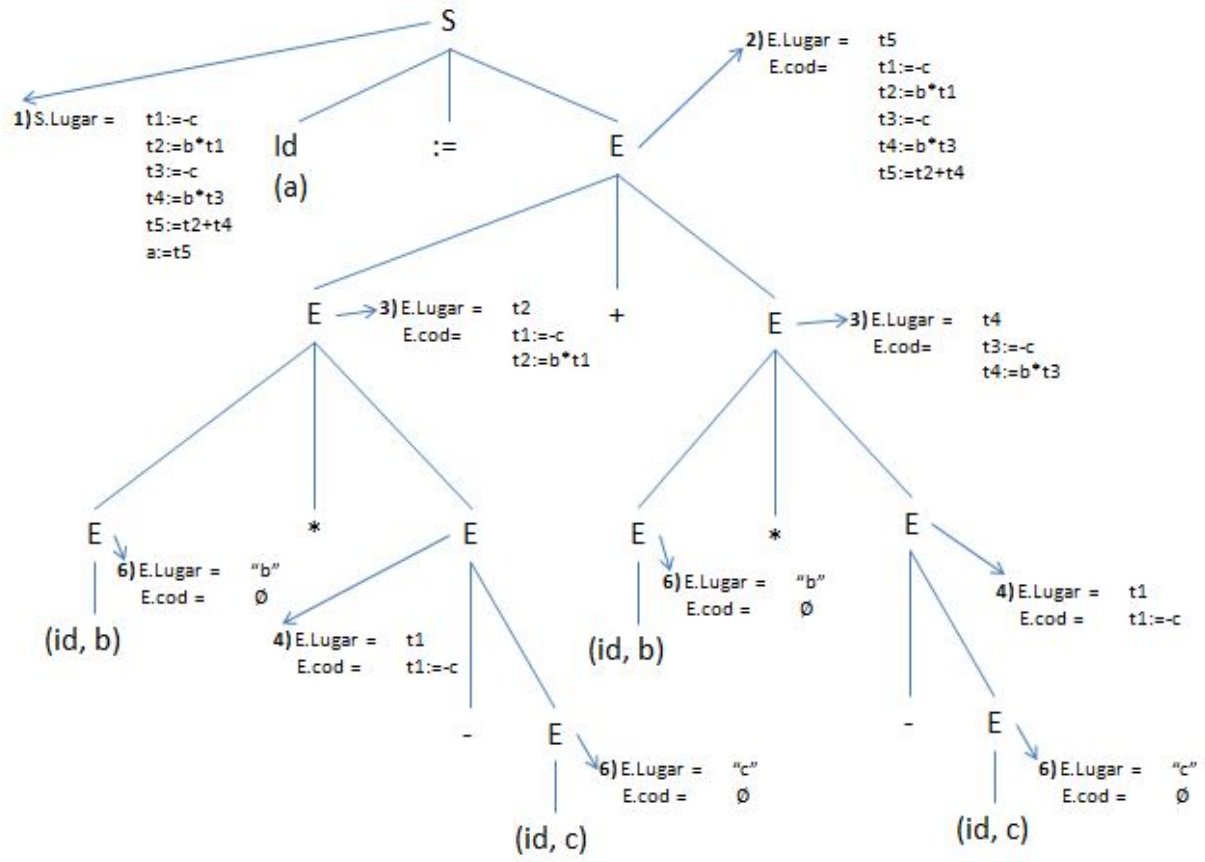
1. S.cod := … (Únicamente ya que no tiene S.lugar)
2. E.lugar := nuevoTemp(...
   E.cod := E_1.cod || E_2.cod || gen (... '+'...)
6. E.lugar := BuscaLugarTS(id.entrada)

Solución ejercicio 1

(Utilizando notacion DDS)

1. **S → id := E →** S.cod := E.cod || gen (BuscaLugarTS(id.entrada), ":=", E.lugar);
2. **E → E1 + E2** → E.lugar := NuevoTemp();
   E.cod := E1.cod || E2.cod || gen (E.lugar, ":=", E1.lugar, "+" E2.lugar);
3. **E → E1 * E2** → E.lugar := NuevoTemp();
   E.cod := E1.cod || E2.cod || gen (E.lugar, ":=", E1.lugar, "*" E2.lugar);
4. **E → -E1** → E.lugar := NuevoTemp();
   E.cod := E1.cod || gen (E.lugar, ":=", "-", E1.lugar);
5. **E → (E1)** → E.lugar := E1.lugar;
   E.cod := E1.cod;
6. **E → id** → E.lugar := BuscaLugarTS(id.entrada);
   E.cod := ;

*Probamos que la expresión **a:=b*-c+b*-c** genera el código esperado mediante un árbol anotado (descendente, en este caso):*



La traducción de la entrada es, por tanto:

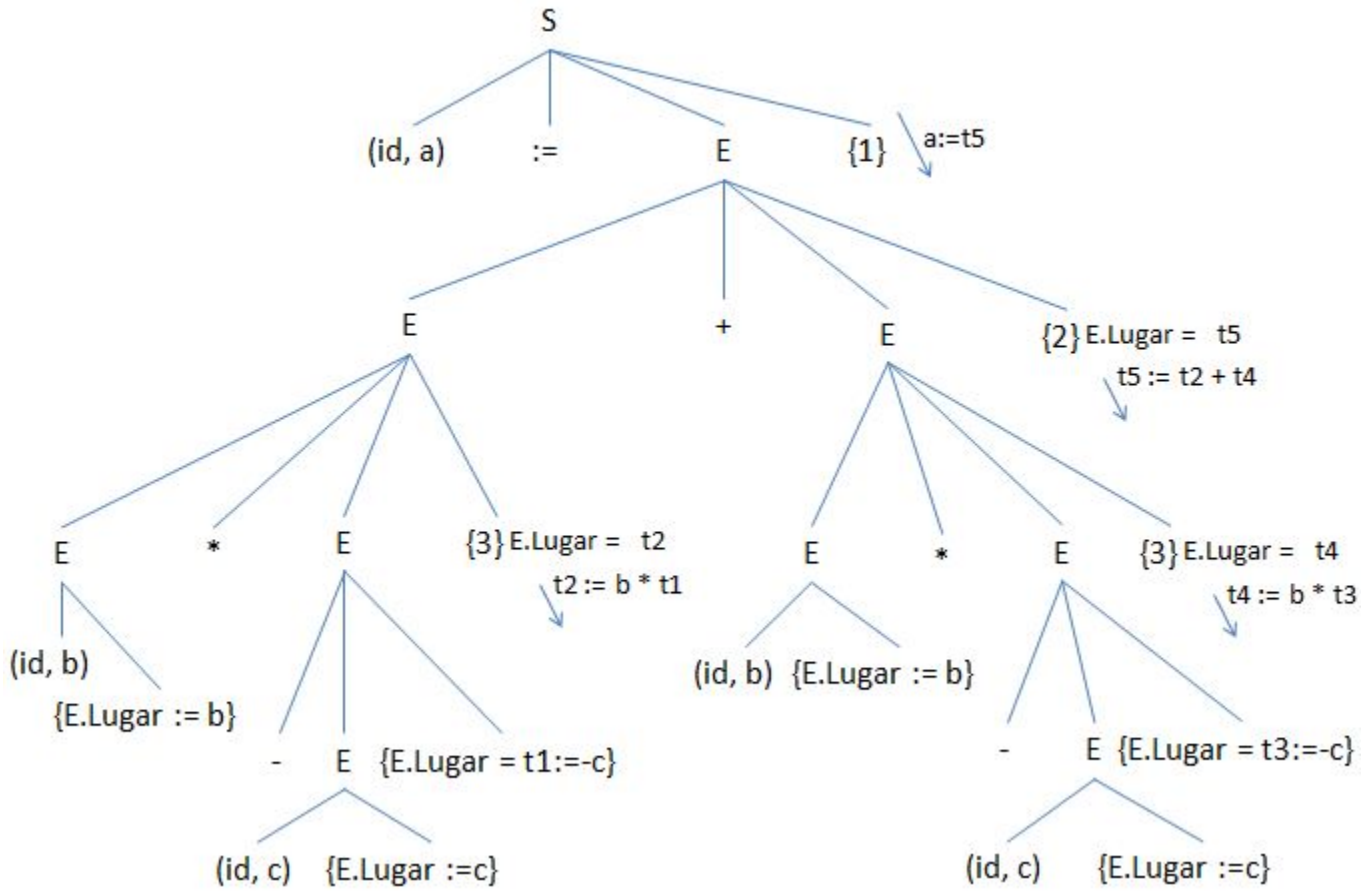
1. *t1 := -c*
2. *t2 := b*t1*

3. *t3 := -c*
4. *t4 := b*t3*
5. *t5 := t2 + t4*
6. *a := t5*

(Utilizando notacion EdT)

1. **S → id := E** {emite (BuscaLugarTS(id.entrada), ":=", E.lugar);}
2. **E → E1 + E2** { E.lugar := NuevoTemp();
   emite (E.lugar, ":=", E1.lugar, "+", E2.lugar);}
3. **E → E1 * E2** { E.lugar := NuevoTemp();
   emite (E.lugar, ":=", E1.lugar, "+", E2.lugar);}
4. **E → -E1 →** { E.lugar := NuevoTemp();
   emite(E.lugar, ":=", "-", E1.lugar);}
5. **E → (E1) →** { E.lugar := E1.lugar;}
6. **E → id →** { E.lugar := BuscaLugarTS(id.entrada);}

*Probamos que la expresión **a:=b*-c+b*-c** genera el codigo esperado:*



La traducción de la entrada es, por tanto:

1. *t1 := -c*
2. *t2 := b*t1*
3. *t3 := -c*
4. *t4 := b*t3*
5. *t5 := t2 + t4*
6. *a := t5*

2.2. Diferencias entre DDS y EDT

DDS	**EDT**
.cod	∅
R. Semanticas y R. Sintacticas	Reglas con las acciones entre { y }
gen	emite

2.3. Juego de Instrucciones del Código 3-d

- x:= y op z | Básicos
- x := op y |
- x := y |
- goto Etiq | Salto
- if x relop y goto Etiq |
- Param x | Paso de parámetros
- Call p (*t:= Call p)* | Llamada al procedimiento P con n parámetros.
- x := y[i] | Operaciones con vectores
- x[i] := y |
- x := &y | Direcciones
- x := *y |
- *x := y |

2.3.1. Trabajando con Expresiones Booleanas

En un lenguaje de programación también disponemos de herramientas lógicas que, por supuesto, podemos añadir a nuestro modelo.
Existen dos formas de trabajar con valores booleanos:

- Por Representacion Numérica (manejamos el Atributo E.lugar): Cada valor lógico lo representamos con un valor numérico para poder hacer comparaciones numéricas:

    - 1 V
    - 0 F
- Por Control de Flujo (Manejamos los Atributos E.verdad y E.falso, que son Atributos Heredados): Podemos tratar los valores lógicos como etiquetas a los que se puede saltar en caso de falso o en caso de verdadero:
    - E.verdad
    - E.falso

Ejemplo

Dada la expresión:

$$a\ or\ b\ and\ not\ c$$

Si existen (or, and, not) como operaciones en 3-d:

- t1 := not c
- t2 := b and t1
- t3 := a or t2

Ejercicio 2

Diseñar el Generador de Código Intermedio mediante Definición Dirigida por la Sintaxis (DDS) para esta gramática y que obtenga como salida el código de 3 direcciones (3-d):

1. E → E1 or E2
2. E → E1 and E2
3. E → not E1
4. E → (E1)
5. E → id1 oprel id2
6. E → true
7. E → false
8. E → id

Solución ejercicio 2

(Con representacion Numérica)

1. **E → E1 or E2 →** E.lugar := NuevoTemp();
   E.cod := E1.cod || E.cod || gen (E.lugar, ":=", E1.lugar, "or", E2.lugar);
2. **E → E1 and E2 →** E.lugar := NuevoTemp();
   E.cod := E1.cod || E.cod || gen (E.lugar, ":=", E1.lugar, "and", E2.lugar);
3. **E → not E1 →** E.lugar := NuevoTemp();
   E.cod := E1.cod || gen (E.lugar, ":=", "not",E1.lugar);
4. **E → (E1) →** E.lugar := E1.lugar;
   E.cod := E1.cod;

5. **E → id1 oprel id2** → E.lugar := NuevoTemp();
   E.cod := gen("if", id1.lugar, oprel.op, id2.lugar, "goto", e_i +3); ||
   gen (E.lugar , ":=", "0"); ||
   gen ("goto", e_i +2); ||
   gen (E.lugar , ":=", "1");
6. **E → true** → E.lugar := NuevoTemp();
   E.cod := gen (E.lugar, ":=", "1");
7. **E → false** → E.lugar := NuevoTemp();
   E.cod := gen (E.lugar, ":=", "0");
8. **E → id** → No vamos a verla

Sabiendo que no existen los operadores and, or y not, rehacer el diseño

1. **E → E1 or E2** → E.lugar := NuevoTemp();
   E.cod := E1.cod || E2.cod ||
   gen ("if", E1.lugar, ":=", "1", "goto", est_ins + 4); ||
   gen ("if", E2.lugar, ":=", "1", "goto", est_ins + 3); ||
   gen (E1.lugar, ":=", "0"); ||
   gen ("goto", est_ins+2); ||
   gen (E1.lugar, ":=", "1"); ||
2. **E → E1 and E2** → E.lugar := NuevoTemp();
   E.cod := E1.cod || E2.cod ||
   gen ("if", E1.lugar, ":=", "0", "goto", est_ins + 4); ||
   gen ("if", E2.lugar, ":=", "0", "goto", est_ins + 3); ||
   gen (E1.lugar, ":=", "1"); ||
   gen ("goto", est_ins+2); ||
   gen (E1.lugar, ":=", "0"); ||
3. **E → not E1** → E.lugar := NuevoTemp();
   E.cod := E1.cod ||
   gen ("if", E1.lugar, ":=", "0", "goto", est_ins + 3); ||
   gen (E1.lugar, ":=", "0"); ||
   gen ("goto", est_ins+2); ||
   gen (E1.lugar, ":=", "1"); ||

2.3.1.1. Expresiones lógicas por representación numérica

Expresiones lógicas por representación numérica → asigna 1 a cierto y 0 a falso.

Diseñar el GCI (que genere instrucciones 3-d) mediante una DDS, manejando las expresiones lógicas por representación numérica

1. S → if E then S_1

2. S → if E then S_1 else S_2
3. S → while E do S_1

Primera aproximación:

1 . S.cod := E.cod || gen (‘if’, E.lugar, ‘=’, ‘0’, ‘goto’, S.despues) ||
S_1.cod || gen(S.despues, ‘:’)

Alternativa más eficiente:

1. S_1.siguiente := S.siguiente [Una etiqueta que marca el final del bloque if-then]
S.cod := E.cod || gen (‘if’, E.lugar, ‘=’, ‘0’, ‘goto’, S.siguiente) ||
S_1.cod

Con ‘else’ el tratamiento es similar, solo que añadiremos una etiqueta local para marcar el inicio del bloque:

2. S_1.siguiente := S.siguiente
S_2.siguiente := S.siguiente
S.else := nuevaetiq

S.cod := E.cod || gen(‘if’, E.lugar, ‘=’, ‘0’ , ‘goto’, S.else) ||
S_1.cod || gen(‘goto’, S.siguiente) || gen (S.else, ‘:’) || S_2.cod

3. S.inicio := nuevetiq
S_1.siguiente := S.inicio
S.cod := gen(S.inicio, ‘;’) || E.cod || gen (‘if’, E.lugar,’=’, ‘0’, ’goto’, S.siguiente) || S_1.cod
|| gen (‘goto’, S.inicio)

Representado en esquema de traducción:

1. **S** → **if E {** emite (‘if’, E.lugar, ‘=’, ‘0’, ‘goto’, S.siguiente) **} then {** S_1.siguiente := S.siguiente **} S1**

2. **S** → **if E {** S.else := nuevaetiq ; emite (‘if’ , E.lugar, ‘=’, ‘0’, ‘goto’, S.else) **} then {** S_1.siguiente := S.siguiente **} S1 {** emite (‘goto’, S.siguiente); emite (S.else, ‘:’) **} else {** S_2.siguiente := S.siguiente **} S_2**

3. **S** → **{** S.inicio := nuevaEtiq ; emite (S.inicio, ‘;’) **} while E {** emite (‘if’, E.lugar, ‘=’, ‘0’, ‘goto’, S.siguiente) **} do [** S_1.siguiente = S.inicio **} S_1 {** emite (‘goto’, S.inicio) **}**

2.3.1.2. Expresiones lógica por control de flujo

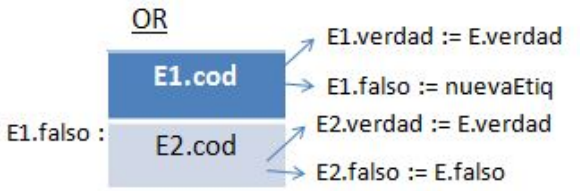
Añadimos los atributos E.verdad y E.falso que son atributos heredados y que indican la etiqueta

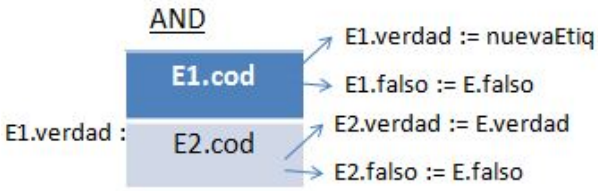
a la que se salta si la expresión es cierta o es falsa:

- .verdad → si es cierta.
- .falso → si no es cierta.

Dada la gramática:

1. E → E1 or E2
2. E → E1 and E2
3. E → not E1
4. E → (E1)
5. E → id1 oprel id2
6. E → true
7. E → false
8. S → if E then S1
9. S → if E then S1 else S2
10. S → while E do S1





1. E1.verdad := E.verdad
   E1.falso := nuevaetiq
   E2.verdad := E.verdad
   E2.falso : E.falso
   E.cod := E1.cod || gen(E1.falso, ':') || E2.cod

2. E1.verdad := nuevaetiq
   E1.falso := E.falso
   E2.verdad := E.verdad
   E2.flaso : E.falso
   E.cod := E1.cod || gen(E1.verdad, ':') || E2.cod

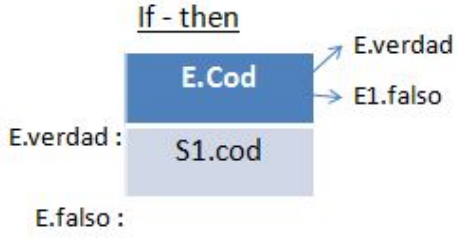
3. E1.verdad := E. falso
   E1.falso := E.verdad
   E.cod := E1.cod

4. E1.verdad := E. verdad
   E1.falso := E.falso
   E.cod := E1.cod

5. id1.lugar := BuscarLugarTS(id1.entrada)
   id2.lugar := BuscarLugarTS(id2.entrada)
   E.cod := gen ('if', id1.lugar, oprel.op, id2.lugar , 'goto', E.verdad) || gen ('goto', E.false)

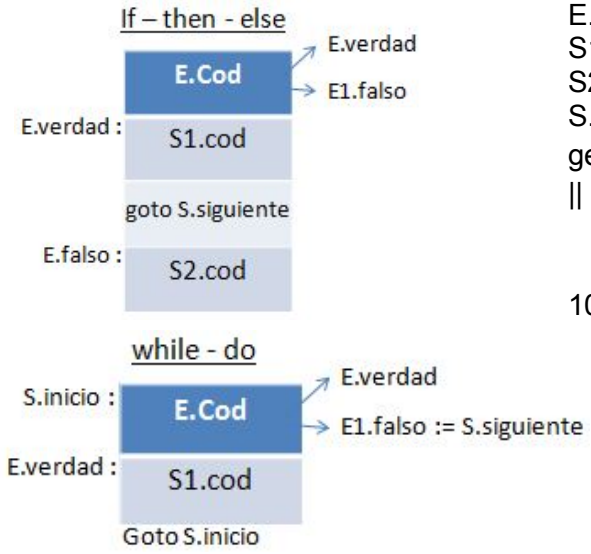
6. E.cod := gen ('goto', E.verdad)

7. E.cod := gen ('goto', E.falso)



8. E.verdad := nuevaetiq

E.falso := S.sig
S1.sig := S.sig
S.cod := E.cod || gen (E.verdad, ‘:’) || S1.cod



9. E.verdad := nuevaetiq
E.falso := nuevaetiq
S1.sig := S.sig
S2.sig := S.sig
S.cod := E.cod || gen (E.verdad ‘:’) || S1.cod |
gen(‘goto’, S.sig) || gen(E.falso, ‘:’)
|| S2.cod

10. E.verdad := nuevaetiq
E.falso := S.sig
S1.sig := S.inicio
S.cod := gen(S.inicio, ‘:’) || E.cod ||
gen(E.verdad, ‘:’) || S1.cod ||
gen(‘goto’, S.inicio)

Ejercicio 3

Imaginamos un lenguaje que permita:

$(a+b)<c$

$(a<b)+(b<a)$

El lenguaje tiene enteros y lógicos, la suma admite cualquier combiandción. Se pide el diseño del GCI (código 3-d) mediante DDS y lógicos por control de flujo:

…
E → E1 + E2
…

Solución ejercicio 3

1. if E1.tipo = entero AND E2.tipo = entero
then [E.tipo := entero]
E.lugar := nuevotemp;
E.cod := E1.cod || E2.cod || gen(E.lugar, ‘:=’, E1.lugar, ‘+’, E2.lugar)

else if E1.tipo = entero AND E2.tipo = lógico

```
then    [E.tipo := entero]
        E.lugar := nuevotemp
        E2.verdad := nuevaetiq
        E2.falso := nuevaetiq
        E.cod := E1.cod || E2.cod || gen (E2.verdad, ':', E.lugar, ':=', E1.lugar, '+', '1' ) ||
                gen ('goto', est_ins +2) || gen (E2.falso, ':', E.lugar, ':=', E1.lugar )

else if E1.tipo = lógico AND E2.tipo = entero
then    [E.tipo := entero]
        E.lugar := nuevotemp
        E2.verdad := nuevaetiq
        E2.falso := nuevaetiq
        E.cod := E1.cod || E2.cod || gen (E.1verdad, ':', E.lugar, ':=', E2.lugar, '+', '1' ) ||
                gen ('goto', est_ins +2) || gen (E1.falso, ':', E.lugar, ':=', E2.lugar )

else if E1.tipo = lógico AND E2.tipo = lógico
then    [E.tipo := entero]
        E.lugar := nuevotemp
        E1.verdad := nuevaetiq
        E1.falso := nuevaetiq
        E2.verdad := nuevaetiq
        E2.falso := nuevaetiq
        E.cod :=E1.cod || E2.cod

        [por hacer!]

else
        E.tipo := tipo_error
```

Ejercicio 4

Podemos complicar el ejercicio anterior con enteros y reales, con conversión implícita de tipos:
entero + entero → entero
real + real → real
entero + real → intoReal(entero) + real = real

…
E → E1 + E2
…

Solucion ejercicio 4

Como esquema de traducción:

E → E1 + E2 { if E1.tipo = E2.tipo = entero
then E.tipo := entero
E.lugar := nuevoTemp
emite (E.lugar, ':=', E1.lugar, '+ ent', E2.lugar)

else if E1.tipo = E2.tipo = real
then E.tipo := real
E.lugar := nuevoTemp
emite (E.lugar, ':=', E1.lugar, '+ real', E2.lugar)

else if E1.tipo = entero AND E2.tipo = real
then E.tipo := real
t := nuevoTemp
emite (t, ':=', ' intoReal', E1.lugar)
emite (E.lugar, ':=', E1.lugar, '+ real', E2.lugar)

else if E1.tipo = real AND E2.tipo = entero
then E.tipo := real
t := nuevoTemp
emite (t, ':=', 'intoReal', E2.lugar)
emite (E.lugar, ':=', E1.lugar, + real', E2.lugar)
else
E.tipo := tipo_error
}

Ejercicio 5

De un lenguaje se extrae un fragmento de su gramática. Se pide GCI mediante esquema de traducción, mediante representación numérica y por control de flujo. En el código intermedio no tiene operadores lógicos.

E --> id / NOT E / E NAND E

Solución Ejercicio 5

Por Representación numérica

(se presupone que todos los identificadores son de tipo Boolean, y por tanto, no realizamos comprobacion de tipos)

E → id { E.lugar := BuscaLugarTS(id.entrada) **}**

E → NOT E1 { E.lugar := nuevoTemp
emite(E.lugar, ':=', '1', '-', E1.lugar) **}**

E → E1 NAND E2 { E.lugar := nuevoTemp ;
emite (if , E1.lugar, '=', 0, 'goto', ei+4) ;
emite(if , E2.lugar, '=', 0, 'goto', ei+3);
emite(e.lugar, ':=', 0);
emite ('goto', ei+2);
emite (E.lugar, ':=', 1)
}

Por control de flujo

E→id { emite ('if', BuscaLugarTS(id.entrada), '=', '1', 'goto', E.verdad)
emite ('goto', E.falso)
}

E → not { E1.verdad := E.falso
E1.falso := E.verdad
} E1

E → { E1.verdad := nuevaEtiq
E1.falso := E.verdad **} E1 nand**
{ emite (E1.verdad, ':')
E2.verdad := E.falso
E2.falso := E.verdad **} E2**

Nota importante:
si tenemos enteros y booleanos, los booleanos podrán representarse mediante control de flujo, pero los enteros deberán estar representados con representación numérica (suponiendo siempre que existe conversión de tipos). En la primera sentencia:

E → id { [además] E.tipo := buscaTipoTS(id.entrada) **}**

En el resto habría que evaluar que tipo tenemos y, si no hay conversión de tipos, no hacer nada si es entero y la evaluación lógica si es tipo lógico:

```
if E1.tipo = E2.tipo = lógico then
        E.tipo = lógico
        * y evaluar la operación
else
        E.tipo = tipoError
```

Todo cambia cuando trabajamos por control de flujo:

E → **id {** id.tipo := BuscaTipoTS(id.entrada)
if id.tipo = lógico then
emite (':f')
emite ('goto')
E.tipo := lógico
else if id.tipo = entero then
E.lugar := BuscaLugarTS(id.entrada)
E.tipo := entero **}**

2.4. Gestión de errores durante la generación de código

Una vez detectado el error, podemos:

- Avisamos y paramos.
- Avisamos y seguimos analizando, pero no generando código.
- Avisamos, corregimos y continuamos.

Tipos de errores:

- Léxicos, sintácticos y semánticos: Errores en la entrada, detectados por el compilador.
- Errores del compilador: por falta de memoria, overflow...
- Errores de ejecución: por ejemplo x/y y=0.s..

Durante la fase de análisis se pueden detectar errores que el gestor de errores se encarga de analizar. En el GCI nos interesa fijarnos en los errores semánticos (sobre todo de tipos) ya que, si se producen, no nos interesa continuar generando código.

Ejercicio 6

(junio 2008)

S → id := E
E → id / E ← E / E & E

LI = Código 3-d. No dispone de operadores lógicos. Tipos enteros y lógicos, sin conversión.

Tabla de verdad

	V	V	F	F
	V	F	V	F
←	V	V	F	V
&	V	F	F	F

Se pide:

1. (EDT) A. Sem. y GCI log. por Control de flujo
2. (DDS) GCI log. por Representacion Numérica

Solución 1

```
S → id := {     E.verdad := nuevaEtiq
                E.falso := nuevaEtiq  }
      E {       id.tipo := BuscarTipoTS(id.entrada) ; id.lugar := BuscaLugarTS(id.entrada)
                S.tipo := if id.tipo = E.tipo AND NOT tipo_error then
                        tipo_ok
                else
                        tipo_error
                [Aquí podríamos comprobar si S.tipo = tipo_OK y si no lo es no entrar en la
                siguiente sentencia if-then]
                if E.tipo = entero then
                        emite (e.lugar, ':=', E.lugar)
                else if E.tipo = lógico then
                        emite (E.verdad, ':')
                        emite (id.lugar, ':=', "1")
                        emite ('goto', S.siguiente)
                        emite (id.lugar, ':=', "0")
        }

E → {   E1.verdad := E.verdad
        E1.falso := nuevaEtiq }
    E1 <= {     emite (E1.falso, ':')
                E2.verdad := E.falso
                E2.falso := E.verdad }
    E2 { if  E1.tipo = E2.tipo = lógico then
                E.tipo = lógico
          else
                E.tipo = tipo_error;  }
```

2.3 Juego de instrucciones 3-d (II)

2.3.2 Instrucciones de llamada a funciones

En nuestro juego de instrucciones tenemos tres instrucciones que nos permiten trabajar con llamadas y retornos de funciones:

- Param x

- Call p
- Return y

Ejemplo
D → procedure id (L) → Se ha metido el tipo del subrgrupo en la TS

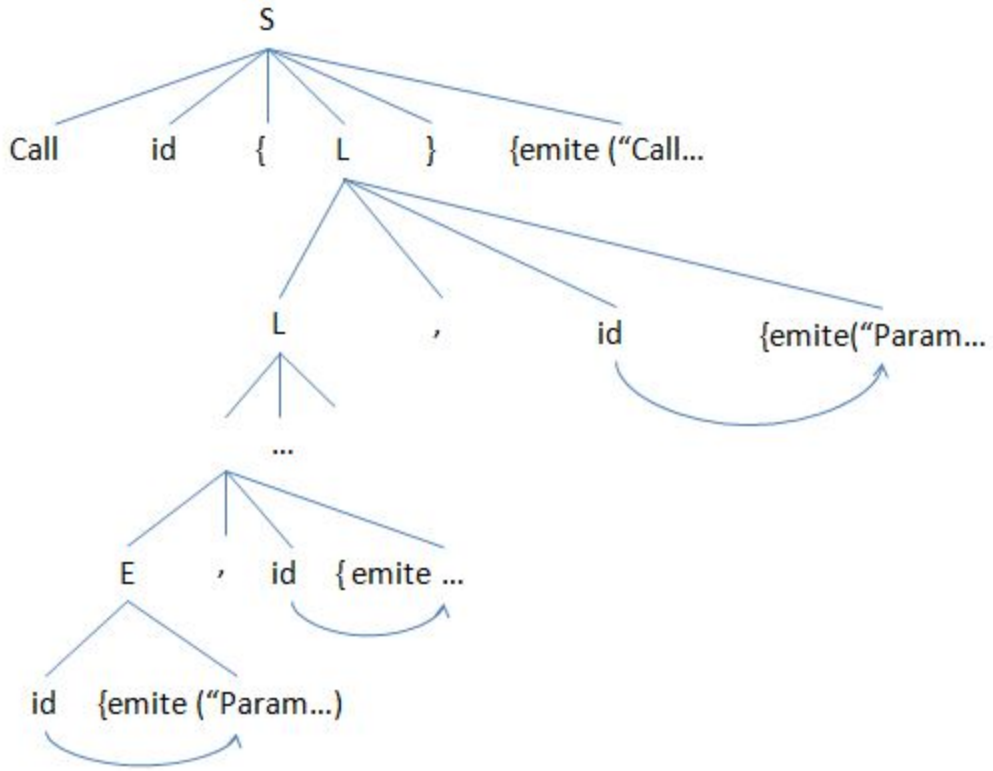
S → call id (L)
→ Comprueba que lo llamamos con el número y el tipo adecuado de parámetros.
L → L, id / id

2.3.2.1 Call y Param

Diseño EdT

S → call id (L) { emite ('call', buscarEtiquetaTS(id.entrada)) **}**
L → L1, id { emite ('Param', BuscarLugarTS(id.entrada)) **}**
L → id { emite ('Param', BUscarLugarTS(id.entrada)) **}**

De esta forma podemos pasar todos los parámetros de la llamada y la llamada en sí:
call suma (a, b, c)



Si los parámetros que se pasan son operaciones no podemos emitirlos como en el ejercicio anterior. Modificando este mismo ejemplo:

S → call id (L)

L → L, E
L → E

Si tratasemos la llamada de la forma anterior el resultado al final sería:

t1 := a + b
Param t1
t2 := c*d
Param t2
Call Et_suma_1
Es decir, la solución se emitirá cuando emitamos la llamada a la función.
No debemos mezclar las partes de código encargadas de la evaluación de las partes que emiten el código 3-d ya que puede darnos problemas más adelante.

La solución correcta será:

S → **{**for i=1 to Long_lista (L.par)
emite('Param', L.par[i])
emite ('call', buscaEtiqTS(id.entrada)) **}**
L → L1, E { L.par := creaLista (L1.par, E.lugar) **}**
L → E { L.par := crearLista(E.lugar) **}**

Por tanto, en la lista L.par se van guardando el E.lugar de cada parámetro y se emiten todo al final, dando como resultado:

t1 := a + b
t2 := c * d
Param t1
Param t2
Call Et_suma_1

Donde la parte de evaluación se separa de la parte de emisión.

Diseño DDS

En DDS es más simple si añadimos dos tipos de .cod:

- .codE que se encargue del código de evaluación
- .codP que se encargue de la generación de parámetros

Sobre el mismo ejemplo:

L → E { L.codE := E.cod
L.codP := gen('Param', E.lugar)}

L → L1, E { L.codE := L1.codE || E.cod
L.codP := L1.codP || gen (‘Param’, E.lugar)}

S → call id (L) { S.cod := L.codE || L.codP || gen(‘call’, BUscaLugarTS(id.entrada)) }

2.3.2.2 Return

S → Return id { S.cod := gen(‘Return’, BuscaLugarTS(id.entrada))
También generamos un return cuando se termine la función para evitar posibles errores.

2.3.3 Instrucciones para el manejo de vectores

Existen dos instrucciones que podemos utilizar para el manejo de vectores:

- x := y[i]
- x[i] := y

2.3.3.1 Dirección

Si tenemos

y[i]

la dirección de este elemnto se calculará con la fórmula:

$$base + (i - l[?][?]mite_{inferior}) * a$$

donde:

- base es la dirección de y.
- i es el número del elemento.
- $límite_{inferior}$ es el límite inferior de un vector (que depende del lenguaje).
- a es el ancho de cada elemento del vector.

Podemos simplificar la fórmula:

$$base + i * a - limite_{inferior} * a$$

donde sabemos que:

$$base - limite_{inferior} * a$$ → es constante.

Para ahorrar trabajo puede añadirse ese valor en la fila de entrada del vector en la TS. Esta constante se calculará a partir del desplazamiento que puede ser igual a la base.

Para una matriz:
Si m(i,j) Si se almacena por filas:

$$base + [(i - limite_{inferior1}) * n + (j - limite_{inferior2})] * a \Rightarrow$$
$$\Rightarrow base - limite_{inferior1} * n * a - limite_{inferior2} * a\ (Constante) + i * n * a + j * a\ (Variable)$$

La parte constante se sitúa en la TS.

Ejemplo

En el Programa fuente tenemos la instrucción:

```
p := m[i]
```

En código intermedio:

```
t1 := m[i]
p := t1
```

No es un cambio demasiado grande. Es a la hora de traducir a código objeto donde debe calcularse su dirección utilizando la constante guardada en la función.

2.3.4 Más instrucciones 3-d

x := &y → Guarda en x la dirección de y. Se utiliza en el paso de parámetros por referencia, por ejemplo.

x := *y
*x := y
Se utilizan para el trabajo con punteros.

Ejercicio (examen jun 2004)

Diseño (mediante EdT)
A. sem
GCI (3-d)

S → A / B / F / Call id (L); / SS
A → id = id; / id = (id + id);
B → repeat M until E > 0;
F → procedure id L {M};
L → L, id / id
M → MA / A
E → id / (E)

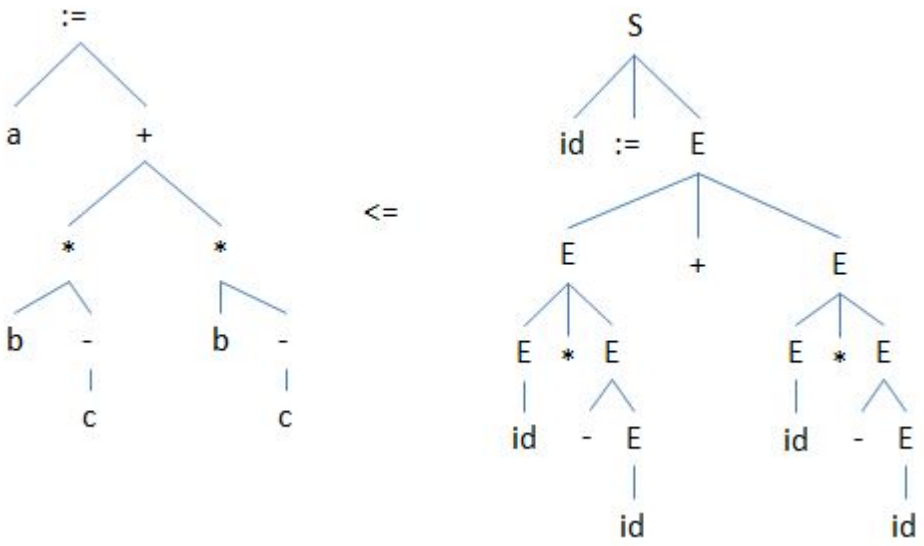
2.5 GCI con otros lenguajes intermedios

2.5.1 Árboles

Dada la siguiente gramática:
1. S → id := E
2. E → E1 + E2
3. E → E1 * E2
4. E → - E1
5. E → (E1)
6. E → id

Se pide crear el árbol sintáctico

resultante.
Por ejemplo: $a := b*-c+b*-c$



Con DDS:

1. S.punt := crea_nodo(':=', crea_nodo_hoja('id', BuscarLugarTS(id.entrada)), E.punt)
2. E.punt := crea_nodo('+', E1.punt, E2.punt)
3. E.punt := crea_nodo('*', E1.punt, E2.punt)
4. E.punt := crea_nodo_unario('-u', E1.punt)
5. E.punt := E1.punt
6. E.punt := crea_nodo_hoja('id', BuscaLugarTS (id.entrada))

2.5.2 Polaca Inversa

Debería dar como resultado: $abc-*bc-*+:=$
Con DDS

1. S.rpn := push(BuscaLugarTS(id.entrada)) || E.rpn || push(':=')
2. E.rpn := E1.rpn || E2.rpn || push('+')
3. E.rpn := E1.rpn || E2.rpn || push('*')
4. E.rpn := E1.rpn || push ('-u')
5. E.rpn := E1.rpn
6. E.rpn := push (BuscarLugarTS(id.entrada))

Con EdT

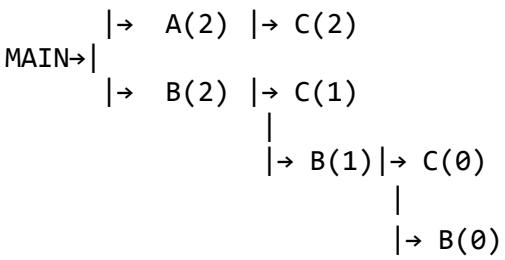
1. S → **id {** push(BuscarLugarTS(id.entrada)) **} := E {** push(':=') **}**
2. E → **E1 + E2 {** push('+') **}**
3. E → **E1 * E2 {** push('*') **}**

4. **E → - E1 {** push(‘-u’) **}**
5. **E → (E1)**
6. **E → id {** push(BuscarLugarTS(id.entrada)) **}**

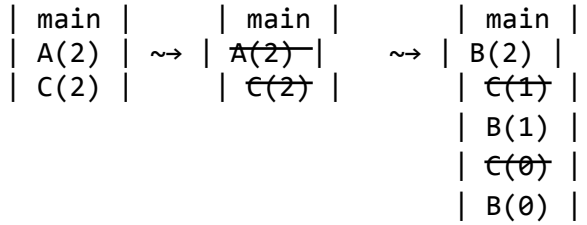
Entorno de Ejecución (Runtime)

Ámbito de los nombres
Activación de un procedimiento
Código del procedimiento → parámetros fromales
Llamada, argumento que se pasan → parámetros actuales
Árbol de activaciones → formado por el nodo principal y cada uno de los nodos son los procedimientos a llamar.

Ejemplo de árbol de activaciones:
[falta código de ejemplo: pedir a Guillermo]



También se puede representar en forma de pila, sin que se pierda ninguna información:



Se usa una pila si:

- El control es secuencial
- Siempre termine el procedimiento llamado antes que el llamante

La variables tienen:

- leftvalue o entorno → dirección de memoria
- rightvalue o estado → valor almacenado en esa dirección.

Organización de la memoria en tiempo de ejecución

- Código del programa [code]
- datos estáticos (variables) [data]
- Árbol de activación → Pila de activación o Pila de control [stack]
- resto de datos dinámicos: heap

Estrategias de asignación:

- Asignación estática: Vale para los casos en el que se conoce el tamaño y el número de todo lo que se necesite almacenar en tiempo de ejecución.
    - No se permitirían llamadas recursivas
    - No se permitirían variables dinámicas.
    - Tiene la ventaja que es muy sencillo y que se puede pueden reutilizar variables en los procedimientos (persistencia).

  Se organizaría en Código del programa y Datos estáticos.
  Ejemplo: Fortran

- Asignación dinámica por pila de Control:
    - Control secuencial sin concurrencia.
    - Lamadas recursivas
    - Variables dinámicas
    - Se podría usar heap, pero sólo para variables dinámicas.

  Se organiza en Código de programa, Datos estáticos y Pila de control (y Heap).
  Ejemplo: Pascal, C ...

- Asignación dinámica mediante Heap:
    - Permite concurrencia
    - Se requiere que cada llamada se de un puntero a la función llamante, para poder saber a dónde volver.

  Se organiza en Código de programa, Datos estáticos y Heap.
  Ejemplo: Cualquier lenguaje concurrente (¿Erlang?)

Registro de activación

Parámetros
Valor devuelto
Variables temporales
Variables locales
Estado Máquina (DR, dirección de retorno) [R1, R2... PC]

Puntero de Acceso: sirve para acceder a variables no locales.
Punteo de Control (o enlace de control): apunta al registro de activación del procedimiento llamante. Necesario para lenguajes concurrentes o cuando se guarde en pila estructuras dinámicas.
